TOWARD HIGH-LEVEL SYNTHESIS OF RELIABLE CIRCUITS THROUGH LOW-COST MODULO SHADOW DATAPATHS

BY

PRANAY VISSA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Associate Professor Deming Chen

ABSTRACT

With transistor dimensions shrinking to the atomic scale, a plethora of new reliability problems presents a barrier to continued Moore's law scaling. Traditional modular redundancy techniques with 2x and 3x area cost eliminate the area reduction benefits of such scaling. In this study, we take a partial redundancy approach to the reliability problem for arithmetic-orientated datapaths by performing lightweight shadow computations in the mod-b space, where b is the base of our modulo residue, for each main computation. We leverage the binding and scheduling flexibility of high-level synthesis to detect control errors through diverse binding and minimize area cost through intelligent checkpoint scheduling and modulo-b reducer sharing. We introduce logic and dataflow optimizations to further reduce cost. We evaluated our technique with 12 high-level synthesis benchmarks from the arithmetic-oriented PolyBench benchmark suite using FPGA emulated netlist-level error injection. When $b = 3$, we observe coverages of 99.2% for stuck-at faults, 99.5% for soft errors, and 99.8% for timing errors with a 25.7% area cost and negligible performance impact. When $b = 5$, we observe coverages of 99.4% for stuck-at faults, 99.8% for soft errors, and 99.9% for timing errors with a 48.5% area cost and negligible performance impact. Leveraging a mean error detection latency of 13.92 and 14.96 cycles, with both mod-3 and mod-5 units respectively (2554x faster than end result check) for soft errors, we also explore a rollback recovery method with an additional area cost of 28.0% for both cases, observing 411x increase in reliability against soft errors.

To my parents, for their love and support.

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my advisor, Associate Professor Deming Chen. His support, motivation, enthusiasm and immense knowledge have been the key to my success in research. His words, "it is time you burnt the midnight oil" pushed me to great extremes and encouraged me to complete great work. I cannot imagine having an advisor who could guide me better towards completing my thesis.

Besides my advisor, I thank Keith A. Campbell for having immense faith in me and helping me through the course of my research. He has led by example and has pushed me to great extremes which eventually led to successful acceptance of my work at a big conference. I always thought of Keith as "my boss" and I would not have completed my thesis if it weren't for his motivation and support.

During my 2 years, I thank my friends from the ES-CAD research group: Yao Chen, Wei Zuo, Ashutosh Dhar, Anand Ramachandran, Yun Heo, Chen-Hsuan Lin, Di He, Zelei Sun, Daniel Chen, Ying Chen, Jong Lim, Ying-Yu Chen, Sitao Huang, Warren Kemmerer and Yi Liang for our stimulating discussions, the long nights we spent before our deadlines and the numerous birthdays we celebrated. The lab would never have been this fun without you guys.

Last but not the least, I thank my parents, Sanjay Vissa and Usha Vissa, for supporting me throughout my life, and my brother Prashant Vissa for providing me assistance in numerous ways. I also thank my extended family: Samir Mitra, Sundari Mitra, Sachin Mitra and Shivani Mitra for being my cheerleaders through my masters.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

HLS	High Level Synthesis
RTL	Register-Transfer Level
TMR	Triple Modular Redundant
DMR	Double Modular Redundant
FPGA	Field-Programmable Gate Array
LLVM	Low Level Virtual Machine
CED	Concurrent Error Detection
VLSI	Very Large Scale Integration
SEC	Statistical Error Compensation
ERC	End Result Check

CHAPTER 1

INTRODUCTION

Moore's law has been defined as a firm ruler in the electronics industry. With demand for more computation density and more performance per watt, there are a huge number of reliability problems which could threaten to end such scaling. With higher chip densities, permanent faults such as burnt-out power supplies become quite common. Transient faults such as particle strikes and intermittent faults such as loose connections have higher probabilities of occurrence. Due to increases in process variations and the challenge of reliably powering billions of transistors, timing errors have also become more common.

Unfortunately, this trend has resulted in increasingly complex hardware that requires high development cost to design. This rapid increase in hardware system complexity has encouraged designers to explore higher levels of abstractions with better productivity than low level RTL [1]. High level Synthesis (HLS) has emerged as a promising way to handle this complexity and reduce development efforts through higher levels of abstraction [2], [3], [4]. Higher abstraction levels also provide synthesis engines with richer information about the behavioral intent of the hardware designer (e.g. algebraic computation expressions, control flow), thus enabling transformations and higher level optimizations which would not be possible at the RTL level.

A traditional approach to error detection in hardware designs is by duplicating each component, also called dual modular redundancy (DMR) [5]. But this approach comes with a 2x area cost that eliminates the area and power reduction benefits of Moore's law scaling. DIVA [6] is another popular technique which uses an extra checker core to verify the correctness of the main core computation and commit only non-faulty results. Concurrent error detection (CED) [7] uses HLS to introduce redundancy at the functional unit level. Although each component is fully duplicated, this technique aims at reducing area and performance overhead through resource sharing. But this

technique can incur at least 75% area cost for simple and small datapaths.

Another approach is time-redundancy, where we re-compute results using the same hardware units to detect errors. [8] uses a time redundancy-based concurrent error detection scheme with diverse binding solutions in its re-computation stage but has performance overheads even though it incurs low area cost. Argus [9] is a prototype processor with a modulo-3 arithmetic checker that can detect up to 98.0% and 98.8% of unmasked transient and permanent errors respectively. Argus has low area (17%) and performance (4%) costs but it is limited to the Von Neumann processor architecture and, to the best of our knowledge, there is no similar work in high-level synthesis that targets application-specific custom logic and accelerator designs.

The traditional approach to reliable hardware is triple modular redundancy (TMR) [10] where 2 additional units are added to the main unit and a majority voting unit. The 3 units perform the same computation and if any of the three units fail, the other two units can correct and mask the fault. Although TMR has a high fault coverage, it has a 3x area cost. [11] integrated modular redundancy into high-level synthesis and presented techniques to increase reliability with cost and performance constraints and decrease cost given reliability constraints, but not both together. New approaches to modular redundancy such as statistical error compensation (SEC) involving pairing an estimator module with unreliable hardware still come with high (50-100%) area cost [12]. Razor [13] is a gate level transformation that adds a shadow latch for each flip-flop to detect timing errors. Although it has a low area and performance overhead (<3%), it is limited to only detecting timing errors. [14] proposes a technique to recover from soft errors but does not perform any error injection experiments and has a passive approach to masking errors whereas we actively detect and correct errors.

In this study, we take a low-cost approach to the error detection problem through modulo-3 arithmetic targeting custom hardware through a fully automated high-level synthesis process enabling rapid development of self-checking hardware. Our innovations are as follows:

1. Intelligent scheduling of intermediate register consistency checks for maximum coverage with minimum consistency checker allocation

2. Pipelining for deferred scheduling of the shadow datapath to reduce area and latency cost and binding diversity to improve fault coverage.

3. Full-coverage handling of mixed arithmetic and non-arithmetic data paths

4. High coverage of 99.42% of unmasked errors for an assortment of three different kinds of fault models with low area overhead.

5. A register-duplication based checkpointing technique to demonstrate the error correction potential of our approach.

6. Previously unexplored area/latency optimizations to modulo-3 functional units through the exploitation of *don't care* cases.

7. An FPGA accelerated, fully automated error injection framework using a gate-netlist transformation to enable accelerated injection for three fault models.

8. The unmasked error detection latency is 4150x faster than existing end result check methods.

This work in this thesis is based on our publication, which is to appear in Design Automation Conference (DAC), June 2015 [15].

CHAPTER 2

BACKGROUND

2.1 High-level Synthesis

High-level synthesis (HLS) is an automated transformation which interprets software and creates custom digital hardware that implements the behavior. HLS typically involves the following steps:

1. **Compilation**: Lowering the code to an intermediate sequence of assembly-like instructions and optimizing those instructions typically through the use of a software compiler framework such as LLVM [16].

2. **Allocation**: Determining the number of physical registers and functional units of each type to allocate.

3. **Scheduling**: Generating a state machine corresponding to the control flow of the instructions and assigning instructions to states.

4. **Binding**: Mapping instructions to physical functional units and variables to physical registers.

5. **RTL Generation**: Generating an RTL description of the state machine and datapath from the scheduling and binding solutions.

2.2 Error Detection through Modular Arithmetic

Given two integers, a (the dividend) and b (the divisor), $a \bmod b$ is the remainder of the Euclidean division of a by b. Given integers x, y, z, b with $b \geq 1$, let $(x', y', z') = (x, y, z) \bmod b$. We can now list the properties of the

mod operator key to our work:

$$x + y = z \implies z' = (x' + y') \bmod b \tag{2.1}$$

$$xy = z \implies z' = x'y' \bmod b \tag{2.2}$$

Suppose the left-hand sides of Equations (2.1) and (2.2) represent some computation performed by a functional unit. Then the computation can be checked by computing x', y', z' and verifying that the right-hand equation holds. Through the use of substitution and noting that $x - y = x + (-1)y$, we can see that this "shadow computation" property holds for arbitrarily complex arithmetic expressions composed of the primitives in Equations (2.1) and (2.2). We can check division operations through shadow multiplications, but we focused on addition and multiplication in our study because they are the most fundamental for datapath design.

The choice of b is important as given A, an n-bit binary encoding of an integer a, we want $a \bmod b$ to be a function of all of the bits in A such that if any bit in A changes, $a \bmod b$ changes. This is true for a signed or unsigned encoding iff $2^i \bmod b \neq 0$ for $0 \leq i < n$ iff b is odd and $b \geq 3$, enabling the detection of any single-bit error. To keep our shadow computation logic as lightweight as possible, we choose $b = 3$ and $b = 5$, the two smallest values which satisfy the above constraint.

2.3 Fault Models

In this study, we consider error detection for three fault models: stuck-at faults, transient errors, and timing errors. Stuck-at faults are the result of fabrication defects that leave a gate output stuck at either 0 or 1 regardless of the input. Transient errors simulate soft errors, which are caused by particle strikes induced by cosmic rays that cause an erroneous state change for a flip-flop. Timing errors are the result of a logic value change failing to propagate along a combinational path from a launch flop to a latch flop before the latch window deadline due to unexpectedly high delays along the path. Such delays can be induced by transistor wear-out, voltage droops, or process variation.

2.4 Aliasing

Aliasing, in this context of study, refers to two inputs that map to the same output through $f(x) = x \bmod b$. For example, $(7 \bmod 3) = (10 \bmod 3) = 1$. Suppose 7 is a correct computation result and 10 is an erroneous computation result. In such a case, the error may not be detected through the shadow computation since the mod-3 residue matches the correct result.

2.5 Zero Masking Problem

Zero masking problem refers to the case when a value in the mod-b space (where b is the base for our modulo residue) is zero. This is common when we multiply a number by a constant, which in the mod-b space evaluates to 0. For example, let us consider a case when $x = 7$ and we multiply it with constant 6. With $b = 3$, $x' = 1$ and $y' = 0$, due to which the result in mod-3 space evaluates to 0. In such a case, some of the upstream logic is optimized out by the optimizer and the input to the subsequent unit is set to constant 0. In such a case, we treat these as mod-3 sinks (Section 4.1) such the errors propagated prior to this have been detected.

CHAPTER 3

SHADOW FUNCTIONAL UNITS

3.1 Mod-3 Functional Units

Mod-3 functional units represent the types of functional units which operate in the mod-3 space. Since only two bits are required to encode 3 possible values in mod-3 space, a simple approach is to use two representations for 0: 00 and 11, which is the approach taken for previous designs of mod-3 functional units. Our key innovation is to ignore the 11 encoding and optimize it as a don't care (**U**).

Thus if either input is the **U** value, then the output doesn't matter as the **U** case will never occur in normal operation. As illustrated in Table 3.1 for the mod-3 adder, there are 9 fixed output cases and 7 *don't care* output cases for each two-input mod-3 unit. Through the use of Karnaugh maps, we optimally exploited these don't cares to find a low area cost design expressed as a sum of products. We verified the optimality of our sum of products solution through exhaustive search of all 4^7 possible *don't care* assignments (i.e. to check for better solutions involving compound gates). Table 3.2 shows the effects of our optimization. For logic synthesis, we implemented our designs in Verilog, used Synopsys Design Compiler 2013-12.sp4 with an ARM 45nm standard cell library, and optimized for minimum area. We measure area in μm^2 and delay in ns.

Table 3.1: Modulo-3 adder functional specification table

value	**encoding**
0	00
1	01
2	10
U	11

$+_3$	**0**	**1**	**2**	**U**
0	0	1	2	X
1	1	2	0	X
2	2	0	1	X
U	X	X	X	X

Table 3.2: Optimization results for shadow mod-3 units

Function	**32-bit unit**		**naive shadow**		**optimized shadow**	
	area	delay	area	delay	area	delay
Add	163	1.30	17.6	0.15	9.30	0.08
Multiply	2381	2.05	10.9	0.08	5.75	0.05

3.2 Mod-3 Constant Functional Units

We also consider an additional class of constant operation units generated by high-level synthesis, units that have a constant as one input. We can think of this constant as "baked-in" to the logic of the unit so that structurally the unit has a single input and a single output. For example, a "+10" constant operation unit takes some value x as input and outputs $x + 10$.

Table 3.3: Shadow mod-3 unit metrics for operation with constant c

Function	**c = 0**		**c = 1**		**c = 2**	
	area	**delay**	**area**	**delay**	**area**	**delay**
Add c	0	0	0.96	0.02	0.96	0.02
Multiply by c	0	0	0	0	0	0

Table 3.3 shows the cost of the constant operation versions of our mod-3 units. Since we can reduce each constant to its mod-3 residue at compile time, there are only three versions of each constant unit. We observe that the operations +0 and x1 have no area cost since they lower to the identity function and x0 lowers to the constant zero for multiplication. As discussed in Section 4.4, such operations are optimized out by our high-level synthesis optimization passes.

With such functional unit optimizations, our method has an even greater area-cost advantage over double or triple modular redundancy for arithmetic datapaths.

3.3 Mod-5 Functional Units

Mod-5 functional units represent the types of functional units which operate in the mod-5 space. Since only three bits are required to encode 5 possible values in mod-5 space, we perform the same optimizations for mod-5 units like we did for mod-3 units (Section 3.1). Since 0, 1, 2, 3 and 4 are the only

Table 3.4: Optimization results for shadow mod-5 units

Function	**32-bit unit**		**naive shadow**		**optimized shadow**	
	area	delay	area	delay	area	delay
Add	163	1.30	89.7	0.76	37.0	0.60
Multiply	2381	2.05	75.0	0.75	35.4	0.80

legal values in mod-5 space, we can have the following representations which allow 5, 6 and 7 to be don't cares.

1. **0**: 000 and 101
2. **1**: 001 and 110
3. **2**: 010 and 111

Since functional units in the mod-5 space operate over larger bits, they will incur larger area when compared to mod-3 units. But since they cover more values, they are less susceptible to aliasing (Section 2.4). Table 3.4 shows the cost of the optimized versions of our mod-5 units.

3.4 Mod-5 Constant Functional Units

Like the mod-3 case, we also consider a class of constant operation units generated by high-level synthesis. Table 3.5 shows the cost of the constant operation versions of our mod-5 units. We observe that the operations +0 and x1 have no area cost since they lower to the identity function and x0 lowers to the constant zero for multiplication. Such operations are optimized out by our high-level synthesis optimization passes.

Table 3.5: Shadow mod-5 unit metrics for operation with constant c

Function	**Add c**		**Multiply by c**	
	area	**delay**	**area**	**delay**
$\mathbf{c = 0}$	0	0	0	0
$\mathbf{c = 1}$	5.42	0.91	0	0
$\mathbf{c = 2}$	4.78	0.95	5.40	0.95
$\mathbf{c = 3}$	6.70	0.88	6.10	0.91
$\mathbf{c = 4}$	5.42	0.95	3.83	0.95

3.5 Modulo-b Reducers

Mod-b reducers are our modulo-b residue computing units, where b is the base of the modulo residue. They are implemented as a tree of mod-b adders with the highest level as a series of normalizers, which normalize the input value to be of width $\lceil \log_2 b \rceil$. The input to the normalizers is a sequence of repeating modular weights. For example when $b = 3$, we have $2^0 \bmod 3 = 1$, $2^1 \bmod 3 = 2$, $2^2 \bmod 3 = 1$, $2^3 \bmod 3 = 2$, etc. Since we see a repeating sequence of 2, the input width to the normalizer is 2. When $b = 5$, we see a repeating sequence of 4 and hence the input width to the normalizer is 4.

When $b = 3$, the normalizers are implemented as mod-3 adders. Thus they can be implemented as a tree of $\lceil \log n/2 \rceil$ stages of modulo-3 adders where n is the input width, similar to the tree approach in [17]. An example reducer for $n = 16$ is illustrated in Figure 3.1. The design works by grouping the input bits into pairs and effectively constructing a base $2^2 = 4$ representation of the input value. Since $4^n \bmod 3 = 1$ for all $n \geq 0$, each base 4 digit has the same weight in mod-3 space and thus we can compute the mod-3 sum of all of the digits in a straightforward tree reduction.

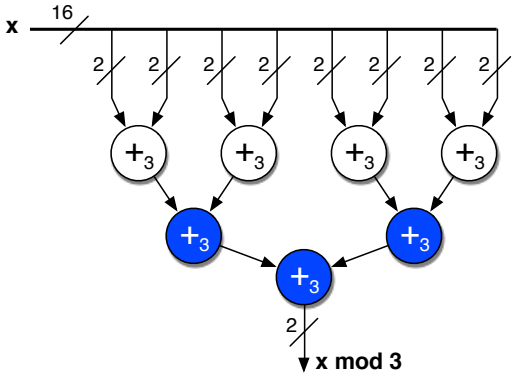


Figure 3.1: Optimized mod-3 reducer topology for a 16-bit unsigned reducer. Optimized mod-3 adders are colored blue.

Since the first stage adders must take all possible values (0, 1, 2, and 3) as inputs, we cannot perform don't care optimizations for those units. But since we design the first stage adders to normalize their output to be 0, 1, or 2, all subsequent stages can optimize the fourth ("3" or U) value as a don't care. To the best of our knowledge, this optimization was not previously explored. With this optimization, we observe a 22-23% area cost reduction and a 23-26% delay reduction compared to [17].

Table 3.6: Optimization results for 32-bit mod-3 reducer

Reducer Type	**[17]**		**ours**	
	area	**delay**	**area**	**delay**
Unsigned	263	0.62	203	0.46
Signed	267	0.66	207	0.51

Thus far, we have assumed that the original datapath uses an unsigned bit encoding for all variables. To modify our reducers to handle a signed (2s complement) variable, we leverage that the only difference between the unsigned and signed (2s complement) encodings is the weight of the most significant bit (MSB). In the unsigned encoding, the MSB has a weight of 2^{n-1} while in the signed encoding, it has a weight of -2^{n-1} where n is the number of bits. Without loss of generality, if we assume n is even, then $2^{n-1} \bmod 3 = 2$ and $-2^{n-1} \bmod 3 = 1$. Since the second most significant bit always has a weight of 1, the insertion of a half-adder is sufficient to normalize the two most significant bits for a signed reducer. Table 3.6 shows the small cost of this extra half-adder.

If $b = 5$, the first stage is a normalizer unit which normalizes 4 bits from the input vector to 3 bits in the mod-5 space, such that it can an input to a tree of mod-5 adders. These mod-5 adders can be optimized because the outputs from the normalizers do not have values outside of the mod-5 space. This optimization helps reduce area cost, as shown in Table 3.7. The additional delay in the optimized versions can be compensated for by using pipeline stages.

Table 3.7: Results for 32-bit mod-5 reducer

Reducer Type	**Unoptimized**		**Optimized**	
	area	**delay**	**area**	**delay**
Unsigned	771	1.09	773	1.48
Signed	409	1.11	408	1.51

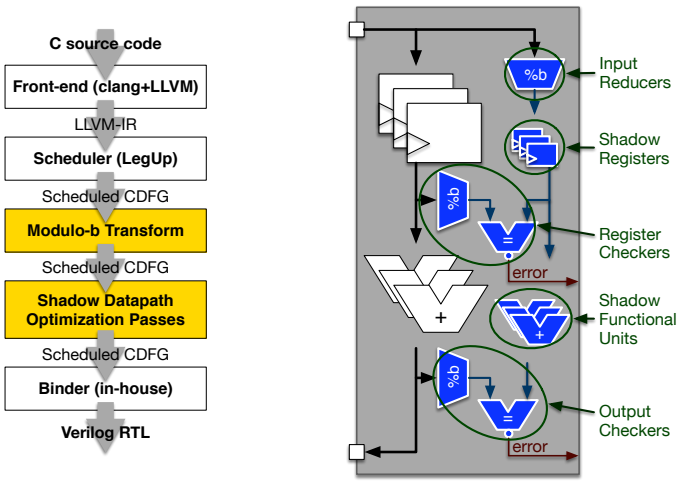
CHAPTER 4

HIGH LEVEL SYNTHESIS TRANSFORMATIONS

Our approach to protecting a hardware design is a series of low-cost shadow datapath high-level synthesis transformations. An overview of how these transformations fit into the high-level synthesis process is illustrated in Figure 4.1a [15]. We perform scheduling with the LegUp high-level synthesis engine [4] and binding with our in-house binding engine. We perform our reliability transformations after scheduling but before binding to insure that the latency of the hardware function does not increase.

Figure 4.1 provides an overview of our basic modulo-3 shadow datapath transformation. Integration of our reliability transformations into the high-level synthesis process is described in Figure 4.1a and the illustration of our core modulo-b transform is described in Figure 4.1b, where b is the base for our modulo residue. The original datapath is colored black/white and the shadow datapath is in blue. For each input port, we add a mod-b reducer to compute the input value mod-b residue, effectively creating a shadow mod-b input. For each arithmetic functional unit (e.g. add, subtract, multiply), we add a corresponding shadow mod-b functional unit (Chapter 3). For each datapath flip-flop, we add a corresponding $\log_2 b$-bit flip-flop to store and propagate the mod-b checksum in a parallel datapath. For each output port, we add a mod-b checker which consists of a reducer and $\log_2 b$-bit equality comparator, which then drives shared error ports. The result is then that each main computation is independently performed in mod-b space as well, and the two results are checked for consistency.

Our transformations operate on a scheduled control/data flow graph. By leveraging the state machine and data flow graph information available in this HLS stage, we can perform transformations and optimizations not possible at the RTL or gate-level stage. In the following subsections, we discuss how we handle mixed arithmetic-nonarithmetic datapaths, the scheduling of intermediate register consistency checks for maximum coverage with opti-



(a) HLS Overview (b) Modulo-b Transform

Figure 4.1: Overview of our method. (a) Integration of our reliability transformations into the high-level synthesis process. (b) Illustration of our core mod-b transform. The original datapath is colored black/white and the shadow datapath is in blue.

mized sharing, pipelining for deferred shadow datapath scheduling to eliminate clock period overhead and lower area cost, and binding diversity between the main and shadow datapaths for improved fault coverage.

4.1 Shadow Datapath Transformations

HLS generated designs involve non-arithmetic components including state machine logic, bitwise operations, and comparators that have single bit outputs. Each non-arithmetic component is duplicated such that such component has a redundant counterpart. However, such units have low area overhead. For example, bitwise operations have very low area cost and shift by constants have zero area costs. We also observe low overheads for duplication of non-arithmetic units (Area and Delay overheads are mentioned in Table 5.1) .

There are a number of cases to deal with when we generate shadow connections for arithmetic and non-arithmetic components, which are illustrated in Figure 4.2. Connections between two duplicate components and between two mod-b components are straightforward: just make connections correspond-

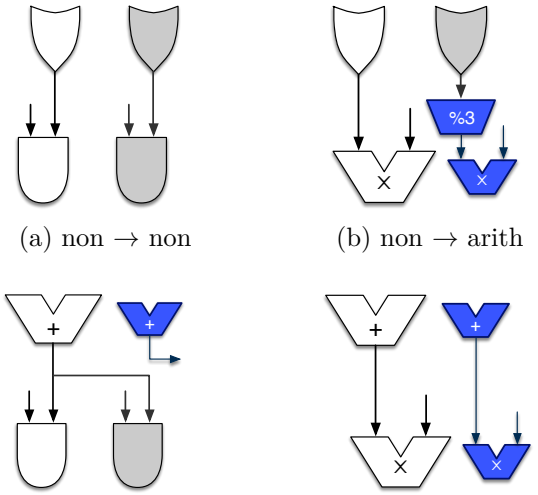


Figure 4.2: Shadow/duplicate connection cases. For each subfigure, the original graph is on the left and the redundant logic is on the right. For the redundant logic, nonarithmetic components ("non") are duplicated with the duplicates in grey. Arithmetic components ("arith") are mod-b shadowed with the shadows in blue. The unit labeled "%3" is a mod-3 reducer.

ing to those in the original datapath (Figures 4.2a and 4.2d). We can connect a duplicate component output (full bit width) to a mod-b component input ($\log_2 b$ bit) through a mod-b reducer (Figure 4.2b, where $b = 3$). Connecting a mod-b component output to a duplicate component input is not possible since information lost in the mod-b reduction cannot be recovered. Thus the duplicate component input is connected to the same output as the original component (Figure 4.2c).

Making connections this way can leave some mod-b components with outputs unconnected, which we call *mod-b sinks*. For example, the mod-b adder in Figure 4.2c may not have a mod-b component to connect to in its fanout. Such mod-b sinks may output an inconsistent mod-b checksum due to an error that occurred in the main datapath, but there would be no way to detect it. Thus we add a mod-b checker for each mod-b sink to insure such errors are detected.

4.2 Register Consistency Check Scheduling

Some errors may be masked in the main datapath (and thus masked in the shadow datapath) before they reach the primary output. Other errors may be unmasked, but undetected due to aliasing (Section 2.4) that occurs in the shadow datapath. To maximize our chances of detecting such errors, we insert checkers on the output of datapath registers, using strategic scheduling of check operations to share as many mod-b reducers as possible.

Compared to the rest of the shadow datapath, reducers are expensive (Tables 3.6 and 3.7). Reducers are scheduled in fixed states for use at output ports and mod-b sinks to produce residues for checkers as well as at input ports to provide a shadow inputs (Figure 4.1b). Intermediate register checkpoints, on the other hand, have flexible scheduling constraints corresponding to their liveness state machine subgraph.

To exploit this flexibility and minimize reducer allocation, we select register liveness intervals that are more than one cycle long and that extend across a basic block boundary (control flow divergence or convergence). For each liveness interval, we attempt to schedule a checkpoint at each *use* (read) of the corresponding SSA variable[1] with the constraint that we cannot schedule more reducers at a state than have been allocated. The intuition behind this method is that we want to catch errors right before they leave a register to go through functional units where they may be masked or aliased. If the checkpoint cannot be scheduled at a state, we attempt to recursively schedule it at each of the state's predecessors.

The core recursive algorithm is listed in Algorithm 1. In the event of a scheduling failure, we allocate an additional reducer and try again until check scheduling succeeds.

4.3 Pipelining for Deferred Shadow Datapath Scheduling

While our mod-b shadow functional units have low latency (Tables 3.2, 3.3, 3.4 and 3.5), our mod-b reducers have high latency (Tables 3.6 and 3.7). In

[1]Single-static assignment variable which is written only once and thus corresponds to one liveness interval for a variable.

Algorithm 1 Core recursive scheduling algorithm

```
function SCHEDULE(var, state)
    if (var, state) has not been visited or scheduled then
        if reducer_count[state] = max_reducers then
            preds ← state predecessors that var is live in
            if preds = ∅ then
                increment max_reducers
                restart scheduling process
            end if
            for each pred in preds do
                schedule(var, pred)
            end for
        else
            schedule check for (var, state)
            increment reducer_count[state]
        end if
    end if
end function
```

addition, the insertion of a mod-b checker on a mod-b sink's corresponding main component can cause severe timing violations if the main component is part of an operation chain. Even if the timing violations are corrected through gate sizing, the area cost can be quite large as 1x transistors are replaced with 4x and 8x transistors to meet timing requirements. Ideally, we want all of the mod-b components to be mapped to 1x gates for minimum area overhead.

Thus our solution is to insert pipeline flip-flops both in front of and behind each mod-b reducer. The shadow datapath schedule is then deferred by 2 cycles, adding 2 cycles of error detection latency in exchange for reduced area cost.

4.4 Shadow Datapath Optimization Passes

Our mod-b transformation can create no-op identity operations and redundant components. This superfluousness motivated us to add a shadow datapath optimization pass to eliminate them as shown in Figure 4.1a which consists of two components:

1. **Constant propagation and identity elimination**: A +6 adder

results in the generation of a +0 mod-3 component, which is an identity. A x6 multiplier evaluates to a constant 0 in mod-3 space, which could then propagate to other operations and make their result evaluable at compile time.

2. **Redundant component elimination**: A x8 and a x11 multiplier both result in the generation of a x2 mod-3 component. If both multipliers are connected to the same input, the second x2 mod-3 component is redundant and can be removed.

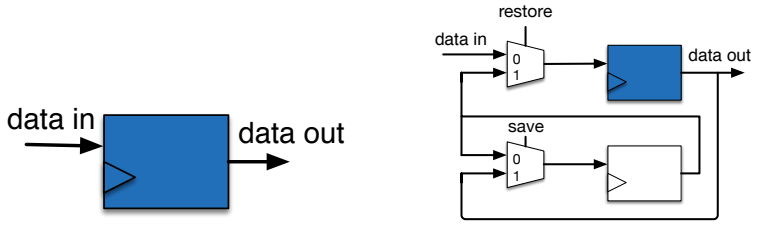
4.5 Diverse Binding

We perform binding of our optimized and scheduled control and data flow graph with our in-house binding engine, which creates diverse (different) binding solutions between the original and duplicate / mod-b datapaths as well as for duplicated non-arithmetic components. Such diverse binding makes it difficult for control errors and stuck-at faults to affect both redundant datapaths in the same way. Further state machine checking is enabled by comparing the state registers of the redundant state machines and using one state machine to control the main datapath and other one to control the duplicate and shadow datapaths. Both the shadow datapath and the duplicate state machine run 2 cycles behind the main computation, so synchronization is not an issue. The binding engine's primary goal is to maximize sharing where profitable for area cost, minimizing the number of reducers allocated.

4.6 Recovery

To enable error recovery for soft errors, we use a checkpoint and recovery register transformation, illustrated in Figure 4.3. For each state and datapath register, we add a duplicate register to store checkpoint data. At regular intervals (configurable), we assert the "save" signal to take a snapshot of the state of each datapath and state register in a corresponding duplicate. Error detection triggers a "restore" signal which recovers the state from the

previously recorded checkpoint, i.e. the cycle where the "save" signal was asserted.



(a) Original flop (b) Transformed flop

Figure 4.3: Flip-flop transformation for soft error recovery.

Our error recovery technique will work for soft errors as long as the error has not made it into the checkpoint snapshot. A checkpoint is corrupted when an error is activated before, but detected after the checkpoint, as shown in Figure 4.4.

Figure 4.4: Checkpoint corruption.

We consider an error to be masked if it does not affect the primary outputs of the generated core or the timing of those outputs. Otherwise, it is an unmasked error. The probability of checkpoint corruption, P_{CC}, is defined as in Equation (4.1), where l is the unmasked error detection latency, P_l is the probability of that particular latency (i.e. $\sum_l P_l = 1$) and CI is the checkpoint interval (configurable). An error is removed if either it is masked to begin with or it is unmasked, detected, and successfully recovered by rolling back to an uncorrupted checkpoint; we formally define the *error removal rate* as the number of removed errors divided by number of total errors, as formalized in Equation (4.2). In this equation, E is the error removal rate; M is the *error masking rate* (defined as number of masked errors divided by number of total errors); and U is the *unmasked error detection rate* (defined as number of unmasked errors detected divided by number of total errors). An error is detected (ED) in a given cycle if an error occurred in that cycle

and it was detected by our detection logic, as formalized in Equation (4.3), where P_{error} stands for the probability of error activation in each cycle and *det* stands for total error detection rate given error activation. *Avg.rollback* is the number of cycles, on average, that we would rollback on detection of an error. Since the rollback length distribution is uniform, the average is approximately half the checkpoint interval (Equation (4.4)). Thus, the average rollback cycle overhead is the product of the average rollback length and the probability of an error being detected in a given cycle (Equation 4.5).

$$P_{\mathrm{CC}} = \sum_{l} P_l \frac{\min(l, \mathrm{CI})}{\mathrm{CI}} \leq \frac{l_{\mathrm{avg}}}{\mathrm{CI}} \tag{4.1}$$

$$\mathrm{E} = \mathrm{M} + \mathrm{U}(1 - P_{\mathrm{CC}}) \tag{4.2}$$

$$\mathrm{ED} = P_{error} \times \mathrm{det.} \tag{4.3}$$

$$\text{Avg. Rollback} = \sum_{r=1}^{\mathrm{CI}} \frac{r}{\mathrm{CI}} = \frac{\mathrm{CI}+1}{2} \tag{4.4}$$

$$\text{Cycle Overhead} = \mathrm{ED} \times \text{Avg. Rollback} \tag{4.5}$$

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 Setup

Our experimental setup is illustrated in Figure 5.1. We performed logic synthesis with Synopsys Design Compiler 2013-12.sp1 with an ARM 45nm standard cell library, and optimized for maximum clock frequency. We evaluated the detection coverage of our approach with error injection enabling netlist transformations which support stuck-at, transient, and timing errors.

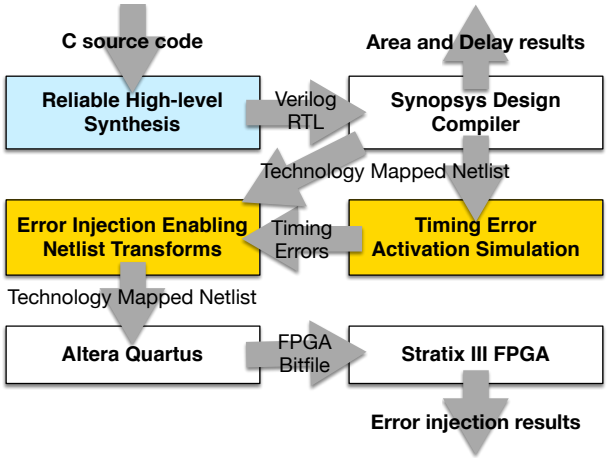


Figure 5.1: Our error detection coverage evaluation framework. Our "reliability-centric" high-level synthesis process is elaborated in Figure 4.1a. Our customized steps are highlighted in yellow.

To inject stuck-at faults, the netlist transform inserts AND (for stuck-at 0) or OR (for stuck-at 1) gates at randomly selected gate outputs. To inject transient errors, we insert XOR gates at the "D" inputs of randomly selected flip-flops. For timing errors, we induce setup time violations by performing timing simulations with a fast clock to collect flop-cycle pairs where timing errors are activated while continuing error-free execution with the use of a

razor flip-flop like transformation, similar to the activation detection method of [18]. Then we pass these flop-cycle pairs as a subset of transient errors to our error injection enabling netlist transformation [15].

To accelerate fault effect evaluation, we map the ASIC netlist to an Altera Stratix III FPGA for emulation. A hardware test driver module mapped to the FPGA communicates with the host system to facilitate thousands of rapid (<1 second each) back-to-back full runs of the design under test, injecting one error from the sample list at a time. As one would expect, stuck-at faults are activated for the duration of the design execution, while transient errors are activated for one cycle.

5.2 Results

We used benchmarks from the PolyBench/C 3.2 benchmark suite and modified the benchmarks to use fixed-point encodings for originally floating-point encoded values as our transformations currently do not support floating-point operations. We implemented fixed point arithmetic with C integer arithmetic operations with shifts for binary point alignment. "Matrix 4x4" is a tiled version of the matrix multiply benchmark that completely unrolls 4×4 tiles to explore performance/area tradeoff.

We synthesized our benchmarks using our method (Chapter 4) and used our experimental setup (Section 5.1). With the addition of our mod-b units, we tested for area overhead and fault coverage, with $b = 3$ and $b = 5$.

To determine the area cost of our error detection approach, we compare the core area of an unprotected baseline benchmark synthesized without our mod-3 shadow datapath transformations against our experimental version synthesized with the mod-3 transforms. Tables (5.1 and 5.2) show the area and clock period overhead for both the detection logic and estimated overhead (through characterization of the hardware in Figure 4.3) for the total logic which includes both detection and recovery. We measure area in μm^2 and delay in ns. Area overhead (area ov.) and clock period overhead (delay ov.) are measured as percentage increase over baseline area.

We observe on average an area cost of 25.7% and 48.5% with detection for mod3 and mod5 units respectively. We estimate 53.7% and 76.4% for both detection and recovery with mod3 and mod-5 units respectively. In-

terestingly, we observe a 9.5% and 22% detection area cost with mod-3 and mod-5 units respectively for the highly parallelized "Matrix 4x4" benchmark, suggesting that lower overheads are achievable in large high-throughput accelerator designs.

Table 5.1: Area and delay overhead results for mod-3 units

Benchmark	**Baseline**		**Detection**		**Total**	
	area	delay	area ov.	delay ov.	area ov.	delay ov.
Atax	13 434	0.89	31.8	-3.1	56.2	1.2
Bicg	13 923	0.90	24.0	-4.1	54.1	0.2
Floyd-Warsh	12 764	0.70	25.2	0.44	55.7	10
Gemm	13 380	0.84	25.8	2.5	51.9	7.1
Gemver	18 855	1.00	28.3	3.3	56.9	7.2
Gesummv	13 230	0.84	26.8	5.0	53.9	9.7
Matrix 4X4	65 258	1.03	9.5	3.9	33.3	7.7
Matrix	11 151	0.80	22.8	1.0	56.3	5.9
Mvt	16 212	0.88	41.6	-1.6	69.2	2.9
Symm	16 943	0.84	23.5	1.1	55.9	5.7
Syr2k	15 183	0.85	26.9	0.8	52.7	5.4
Syrk	13 975	0.89	22.4	0.4	48.2	4.8
Median	13 949	0.86	25.5	1.0	54.9	5.8
Mean	18 763	0.87	25.7	1.1	53.7	5.7

To observe fault coverage, we injected a sampling of 2,000 stuck-at, 10,000 transient and 10,000 timing errors into each synthesized core. The outcome of our fault injection experiments is shown in Tables (5.3) and (5.3).

For unmasked errors, we observe an average stuck-at fault coverage of 99.2%, soft error coverage of 99.5%, and timing error coverage of 99.8% with mod-3 units. To provide some context, Argus, which we consider to be a state of the art error detecting microprocessor, can detect 98.0% of transient errors and 98.8% of stuck-at faults [9]. For unmasked errors with mod-5 units, we observe an average stuck-at fault coverage of 99.4%, soft error coverage of 99.8%, and timing error coverage of 99.9%. There is an increase in fault coverage with mod-5 units, mainly due to lower probabilities of aliasing. Thus with higher bases, we will observe higher fault coverage, but with more area overhead.

It is difficult to make a direct comparison with previous HLS work since high-level synthesis benchmarks with experimental error injection and area cost are quite limited. For reference, Concurrent Error Detection [7] uses

Table 5.2: Area and delay overhead results for mod-5 units

Benchmark	Baseline		Detection		Total	
	area	delay	area ov.	delay ov.	area ov.	delay ov.
Atax	13 434	0.89	52.9	-4.6	77.3	0.2
Bicg	13 923	0.90	43.5	-4.8	73.7	-0.4
Floyd-Warsh	12 764	0.70	54.8	2.1	85.4	7.7
Gemm	13 380	0.84	43.6	2.3	69.7	6.9
Gemver	18 855	1.00	47.6	-0.4	76.1	3.5
Gesummv	13 230	0.84	61.6	0.1	88.7	4.8
Matrix 4X4	65 258	1.03	22.0	1.7	45.8	5.5
Matrix	11 151	0.80	44.8	1.1	78.3	6.0
Mvt	16 212	0.88	75.6	0.2	103	4.9
Symm	16 943	0.84	45.3	1.4	77.6	6.1
Syr2k	15 183	0.85	51.8	-0.5	77.6	4.1
Syrk	13 975	0.89	37.9	-0.1	63.7	4.3
Median	13 949	0.86	46.4	0.3	77.5	4.8
Mean	18 763	0.87	48.5	-0.1	76.4	4.4

HLS to fully duplicate each component but attempts to compensate for area cost through resource sharing and has around 75% area cost for a simple, fully arithmetic datapath which in theory is not susceptible to aliasing.

Figures 5.2 and 5.3 shows the estimated soft error removal rate and roll-back cycle overhead, with mod-3 and mod-5 units respectively, for our error recovery method with checkpoint intervals ranging from 10 to 100k cycles calculated through Equations (4.1) - (4.5). With mod-3 units, the baseline average masking rate of the unmodified designs is 70.1% (indicated by the lower dotted line), and we achieve an total error removal rate (indicated by the "Error Removal Rate" curve) arbitrarily close to the theoretical upper bound (all errors detected are corrected) which is 99.8% (indicated by the upper dotted line).With mod-5 units, the baseline average masking rate of the unmodified designs is 71.8%, and we achieve an total error removal rate arbitrarily close to the theoretical upper bound (all errors detected are corrected) which is 99.9% (indicated by the upper dotted line).

We cannot achieve an error removal rate of 100% as we have a small percentage of undetected, unmasked errors. The 4 parallel lines represent roll-back cycle overheads for different soft errors rates. For reference, [19] reports a worst case error rate of around 10^{-16} errors / cycle for a space environment assuming a clock frequency of 1GHz.

Table 5.3: Fault coverage with mod-3 units

Benchmark	Unmasked			Masked		
	stuck-at	transient	timing	stuck-at	transient	timing
Atax	99.8	99.8	100	73.6	28.7	69.6
Bicg	99.1	97.4	100	77.5	30.6	54.8
Floyd-Warsh	99.7	100	100	74.9	41.5	70.5
Gemm	99.3	100	100	75.4	31.3	79.3
Gemver	99.2	99.9	100	76.7	19.1	82.0
Gesummv	99.9	99.2	100	72.7	38.0	79.9.1
Matrix 4X4	99.2	98.7	99.5	66.7	48.3	67.5
Matrix	99.9	99.9	100	75.6	25.7	48.8
Mvt	96.7	100	100	77.6	16.7	65.0
Symm	99.8	99.3	99.9	78.6	36.4	82.3
Syr2k	98.7	99.7	99.3	73.8	33.4	84.6
Syrk	99.3	100	100	73.5	32.3	82.6
Median	99.3	99.9	100	75.2	31.8	75.0
Mean	99.2	99.5	99.8	74.6	31.9	72.5

What is interesting to observe is the tradeoff between the error removal rate and rollback cycle overhead. Larger checkpoint intervals reduce the chance of checkpoint corruption, resulting in higher error removal rates. At the same time large checkpoint intervals result in larger jumps back in time for each error detection triggered rollback, resulting in larger cycle overheads.

Figures 5.4 and 5.5 shows the soft error detection latency distribution for unmasked errors, masked errors and both. "End Result Check" (ERC) is a basic error detection method involving comparing the benchmark's output

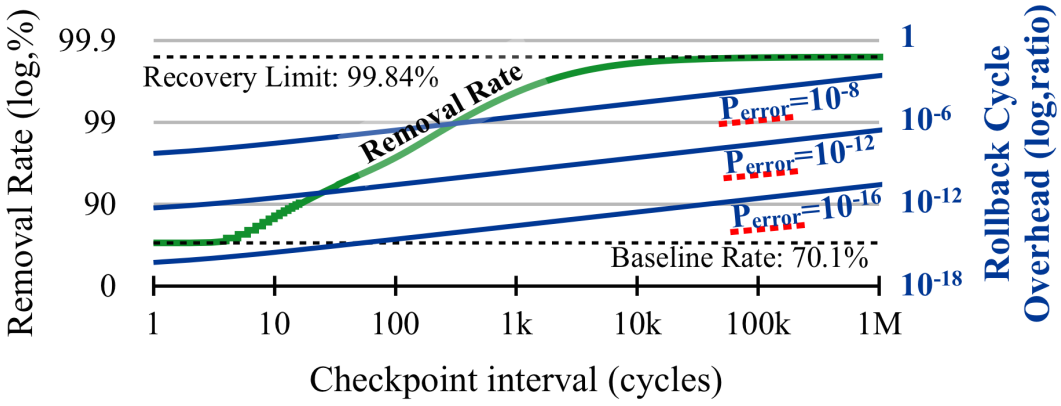


Figure 5.2: Error removal rate and rollback cycle overhead with mod-3 units.

Table 5.4: Fault coverage with mod-5 units

Benchmark	Unmasked			Masked		
	stuck-at	transient	timing	stuck-at	transient	timing
Atax	99.3	99.8	100	78.0	28.1	62.4
Bicg	99.7	99.5	100	82.3	33.3	63.1
Floyd-Warsh	99.9	100	100	74.8	41.9	79.2
Gemm	99.1	100	100	77.3	29.1	74.7
Gemver	99.3	100	100	75.5	19.5	72.4
Gesummv	100	99.5	100	72.6	38.5	60.9
Matrix 4X4	99.1	99.0	99.5	68.1	50.1	80.1
Matrix	100	100	100	77.2	29.2	52.8
Mvt	98.9	100	100	67.7	17.3	61.4
Symm	99.1	99.5	100	78.9	38.2	84.3
Syr2k	98.9	100	100	75.9	33.9	88.2
Syrk	99.1	100	100	76.1	33.9	72.1
Median	99.3	100	100	76.0	33.6	72.5
Mean	99.4	99.8	99.9	75.4	33.0	71.0

with its expected output once execution is complete. With mod-3 units, we observe mean latencies of 11.04, 16.02, 13.92, and 35.5k cycles for unmasked, masked, both and ERC respectively. With mod-5 units, we observe mean latencies of 11.18, 16.28, 14.26, and 35.8k cycles for unmasked, masked, both and ERC respectively. In both cases, we observe an error detection latency improvement of 2554x over the ERC.

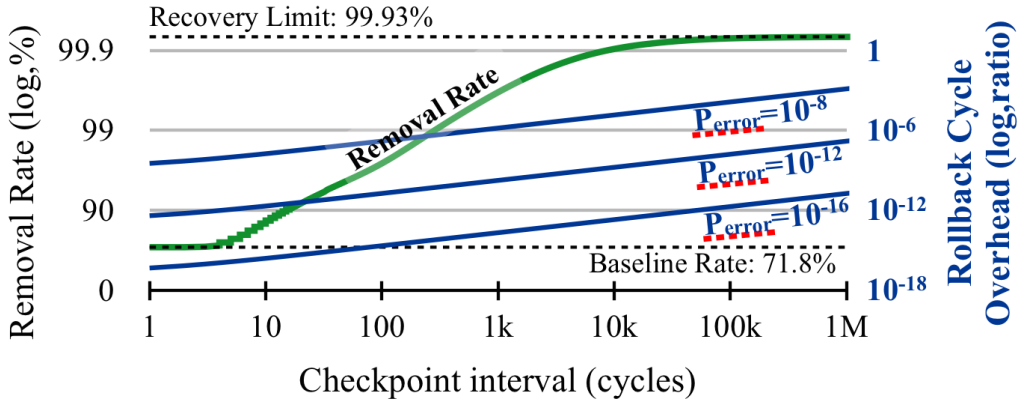


Figure 5.3: Error removal rate and rollback cycle overhead with mod-5 units.

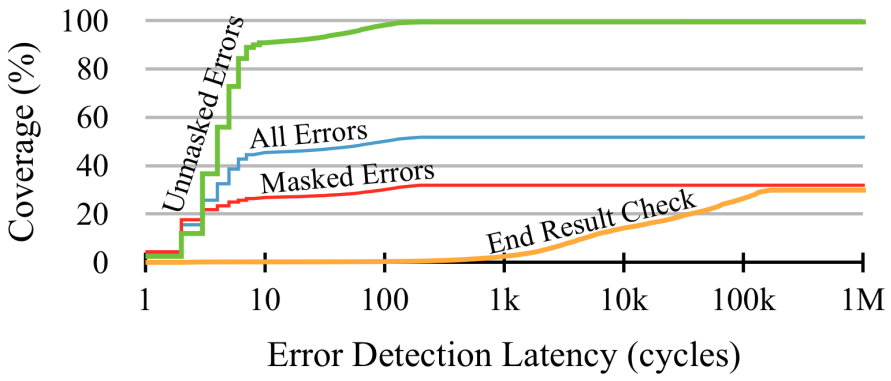


Figure 5.4: Soft error detection latency distribution.

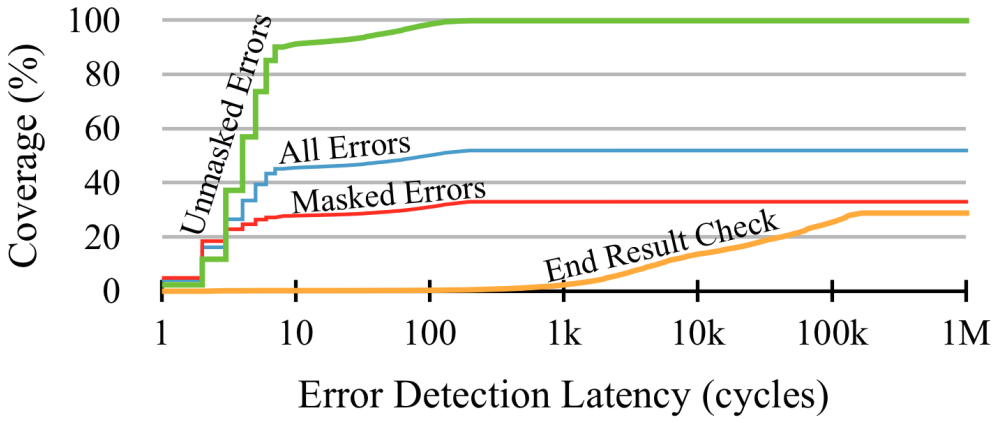


Figure 5.5: Soft error detection latency distribution.

CHAPTER 6

CONCLUSION

We have designed and implemented a fully automated high-level synthesis process to create error detecting cores capable of detecting an average of 99.5% of unmasked errors, with mod-3 functional units, for an assortment of three different kinds of fault models with negligible delay cost, 26% area cost, and a detection latency 3302x faster than an end result check. With mod-5 functional units, we observe an area cost of 48.5% and negligible delay cost and are capable of detecting 99.7% of unmasked errors on average. We have taken the first step towards the fully automated generation of low area cost, low development cost reliable hardware through high-level synthesis. We also explored a rollback recovery method for soft errors with an additional area cost of 28% for both, through which we achieve up to a 411x increase in reliability against soft errors. We observe a mean error detection latency of 13.92 and 14.26 cycles for soft errors, with mod-3 and mod-5 units respectively.

Our next steps include adding support for floating-point operations and fixing suck-at faults through fail-over techniques and timing errors through rollback combined with frequency-voltage scaling. We also plan to include mixed-modular datapaths where we include multiple modulo bases and try to improve fault coverage and area cost.

REFERENCES

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, April 2011.

[2] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in *Field-Programmable Technology (FPT), 2011 International Conference on*, Dec 2011, pp. 1–8.

[3] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 18–25, July 2009.

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. on Embedded Computing Systems*, vol. 13, no. 2, Sep. 2013.

[5] J. Tryon, "Quadded logic," *Redundancy Techniques for Computing Systems*, 1962.

[6] T. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design," in *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, 1999, pp. 196–207.

[7] A. Antola, V. Piuri, and M. Sami, "High-level synthesis of data paths with concurrent error detection," in *Defect and Fault Tolerance in VLSI Systems, 1998. Proceedings, 1998 IEEE International Symposium on*, Nov 1998, pp. 292–300.

[8] K. Wu and R. Karri, "Algorithm level recomputing with allocation diversity: a register transfer level time redundancy based concurrent error detection technique," in *Test Conference, 2001. Proceedings. International*, 2001, pp. 221–229.

[9] A. Meixner, M. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, Dec 2007, pp. 210–222.

[10] J. von Neumann, "Probabilistic logics and synthesis of reliable organisms from unreliable components," in *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton University Press, 1956, pp. 43–98.

[11] R. Karri, K. Hogstedt, and A. Orailoglu, "Computer-aided design of fault-tolerant VLSI systems," *Design Test of Computers, IEEE*, vol. 13, no. 3, pp. 88–96, Fall 1996.

[12] E. P. Kim, "Statistical error compensation for robust digital signal processing and machine learning," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2014.

[13] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, Dec 2003, pp. 7–18.

[14] S. Tosun, O. Ozturk, N. Mansouri, E. Arvas, M. Kandemir, Y. Xie, and W.-L. Hung, "An ILP formulation for reliability-oriented high-level synthesis," in *Quality of Electronic Design, 2005. ISQED 2005. Sixth International Symposium on*, March 2005, pp. 364–369.

[15] K. Campbell, P. Vissa, D. Z. Pan, and D. Chen, "High-level synthesis of error detecting cores through low-cost modulo-3 shadow datapaths," in *Proceedings of IEEE/ACM Design Automation Conference (DAC), 2015*, June 2015.

[16] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, March 2004, pp. 75–86.

[17] S. Piestrak, F. Pedron, and O. Senlieys, "VLSI implementation and complexity comparison of residue generators modulo 3," in *Eusipco: European signal processing conference*, 1998, pp. 511–514.

[18] M. Gao, P. Lisherness, K.-T. Cheng, and J.-J. Liou, "On error modeling of electrical bugs for post-silicon timing validation," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, Jan 2012, pp. 701–706.

[19] A. Bogorad, J. Likar, R. Lombardi, S. Stone, and R. Herschitz, "On-orbit error rates of RHBD SRAMs: Comparison of calculation techniques and space environmental models with observed performance," *Nuclear Science, IEEE Transactions on*, vol. 58, no. 6, pp. 2804–2806, Dec 2011.